Qapla Chess GUI – i18n Module (Translator)

This document describes the internationalization (i18n) subsystem implemented in `src/i18n.h` and `src/i18n.cpp`. It explains the `Translator` class, its responsibilities, threading and persistence behavior, and how to use it correctly from the GUI code.

Overview

The i18n system provides:

- Topic-based translation lookup (e.g. "Button", "Tab", "Dialog").
- Language selection via short language codes (e.g. `"eng"`, `"deu"`, `"fra"`).
- Loading of translations from INI-style `.lang` files or, in release builds, from embedded language data.
- Automatic tracking and persistence of missing translation keys.
- Optional debug mode (`QAPLA_DEBUG_I18N`) for live editing of `.lang` files and automatic timestamp management.

All translation access goes through the singleton `QaplaWindows::Translator` or the global helper function `tr()`.

Class: Translator

Namespace: `QaplaWindows`

The `Translator` class is the central i18n component and derives from `QaplaHelpers::Autosavable`. It manages all loaded translations, the active language, and (in debug mode) synchronization of the source `.lang` files.

Lifetime and Singleton Access

- `static Translator& Translator::instance();`
  - Returns the global singleton instance.
  - The instance is lazily constructed on first use (function-local `static`).
- Constructor
  - Registers itself as an `Autosavable` with the file name `"missing_translations.txt"`, backup suffix `".bak"`, and a save interval of 60 seconds.
  - Uses `Autosavable::getConfigDirectory()` as the base directory for the autosave file.
  - Calls `loadFile()` to restore previously recorded missing translations.
  - Registers a callback with `StaticCallbacks::save()` so that `saveFile()` is invoked on global save events.
- Destructor
  - Currently empty; lifetime is managed by the singleton pattern and `Autosavable` infrastructure.

Thread Safety

- All mutable state related to translations and language selection is protected by `languageMutex`.
- Public methods that access or modify translation data (`translate`, `loadLanguageFile`, `addTranslation`, `setLanguageDirectory`, `setLanguageCode`, `getLanguageCode`, `saveData`, `loadData`) acquire a `std::scoped_lock` on this mutex.
- Callers can safely use `Translator` from different GUI contexts as long as they always go through the public API.

Internal Data Structures

- `TopicMap translations;`
  - Type alias: `using TranslationMap = std::unordered_map<std::string, std::string>;`
  - Maps `topic` → (map of `lookupKey` → `translatedValue`).
- `mutable QaplaHelpers::ConfigData missingKeys_;`
  - Stores missing translation keys per topic in an INI-like structure.
  - Persisted via `Autosavable` to `missing_translations.txt`.
- `std::string languageDirectory = "lang";`
  - Base directory name for language files (used mainly in non-embedded scenarios).
- `std::string currentLanguage = "eng";`
  - Holds the active language code.
- `std::vector<std::string> loadedLanguages;`
  - Tracks which languages have already been loaded during the current run.
- `std::unique_ptr<Callback::UnregisterHandle> saveCallbackHandle_;`
  - RAII handle for unregistering the save callback on destruction.
- Debug only (`QAPLA_DEBUG_I18N`):
  - `std::unordered_map<std::string, std::vector<TimestampUpdate>> pendingUpdates_;`
  - Used to defer timestamp updates for translation keys until the next save.

Public API

translate(topic, key)

```
std::string translate(const std::string& topic, const std::string& key);
```

- Normal usage entry point for looking up translations.

- Steps:

    1. Wraps the input `key` in a `TranslationNormalizer` to normalize whitespace/placeholders.
    2. If the normalized key is empty, returns the original `key` unchanged.
    3. Builds a `TranslationKey` from the normalized key and computes a `lookupKey` (stable identifier used in `.lang` files).
    4. Looks up `translations[topic][lookupKey]` under `languageMutex`.

- If the translation **exists**:

    - In debug mode:
        - Calls `markTimestampUpdate` for each built-in language (`deu`, `eng`, `fra`) to update the timestamp on save.
        - Marks the autosavable as modified via `setModified()`.
    - Returns the stored translation text after `TranslationNormalizer::restorePlaceholders()` is applied.

- If the translation **does not exist**:

    - In debug mode (`QAPLA_DEBUG_I18N`):

        - Resolves the source i18n directory via `getI18nSourceDirectory()`.
        - Builds a key string containing the current date via `TranslationKey::getKeyString(currentDate)`.
        - For each language code (`deu`, `eng`, `fra`):
            - Opens the corresponding `*.lang` file.
            - Adds a missing translation entry with the normalized key written in file format (escaped newlines) using `addMissingTranslationToFile()`.
        - Inserts `translations[topic][lookupKey] = normalizedKey` so that subsequent lookups will find it.
        - Logs the addition via `QaplaTester::Logger` at `TraceLevel::info`.
        - Returns the original `key` (the text is effectively used as the translation until files are edited).

    - In release mode (no `QAPLA_DEBUG_I18N`):

        - Converts the normalized key to file format using `toFileFormat()`.
        - Fetches or lazily creates a `Translation` section in `missingKeys_` for the given `topic`.
        - Adds the missing key to that section if it is not already present.
        - Marks the autosavable as modified via `setModified()`.
        - Returns the original `key` unchanged to be displayed as-is.

Important behavioral guarantee: `translate()` never throws exceptions and always returns some string; when no translation is found, it simply returns the original key.

loadLanguageFile(filepath)

```
void loadLanguageFile(const std::string& filepath);
```

- Loads translations from an external `.lang` file at `filepath`.
- Uses `QaplaTester::Logger` to report errors when the file cannot be opened.
- Delegates the actual parsing to `loadLanguageFromStream()`.

addTranslation(topic, key, value)

```
void addTranslation(const std::string& topic,
                    const std::string& key,
                    const std::string& value);
```

- Adds or overrides a translation at runtime.
- Does not perform normalization; the caller must provide the correct lookup key.
- Mainly useful for tests or programmatic overrides.

setLanguageDirectory(directory)

```
void setLanguageDirectory(const std::string& directory);
```

- Sets the base directory for language files (string only; no path validation).
- Intended for configurations where language files are not embedded.

setLanguageCode(language)

```
void setLanguageCode(const std::string& language);
```

- Changes the active language and (re)loads its translations.

- Behavior:

  1. Locks `languageMutex`.
  2. If the requested `language` is already the `currentLanguage` and it is present in `loadedLanguages`, the call returns early (no-op).
  3. Otherwise it updates `currentLanguage`, clears `translations` and `loadedLanguages`, then unlocks the mutex.

- Debug build (`QAPLA_DEBUG_I18N`):

  - Resolves the source `i18n` directory with `getI18nSourceDirectory()`.
  - Attempts to load `<language>.lang` directly from the source tree.
  - On success:
    - Calls `loadLanguageFile()`.
    - Registers the language in `loadedLanguages`.
    - Logs successful loading at `TraceLevel::info`.
  - On failure:

        - Logs a warning that the language file was not found.

- Release build:

    - Tries to load **embedded** language data first using the compiled-in arrays `deu_lang`, `eng_lang`, `fra_lang`.
    - If embedded data is found:
        - Builds a `std::string` from the raw memory block and wraps it in `std::stringstream`.
        - Calls `loadLanguageFromStream()` with this stream while `languageMutex` is held.
        - Registers the language in `loadedLanguages` and logs success.
        - Returns early.
    - If there is **no embedded data** for the language:
        - Builds a file system path: `<configDirectory>/i18n/<language>.lang`.
        - If the file exists, calls `loadLanguageFile()` and registers the language.
        - Otherwise logs a warning that the language file was not found.

getLanguageCode()

```
std::string getLanguageCode() const;
```

- Returns the currently active language code as a copy.
- Protected by `languageMutex`.

toFileFormat(text) / fromFileFormat(text)

```
static std::string toFileFormat(const std::string& text);
static std::string fromFileFormat(const std::string& text);
```

- `toFileFormat`:

    - Escapes newline characters by replacing `"\n"` with the literal sequence `"\\n"`.
    - Used when writing multi-line translations into `.lang` files.

- `fromFileFormat`:

    - Trims both ends of the input using `QaplaHelpers::trim()`.
    - Replaces `"\\n"` sequences with real newline characters.
    - Used when reading translations back from files.

Autosavable Integration

The `Translator` is an `Autosavable` instance and implements:

```
void saveData(std::ofstream& out) override;
void loadData(std::ifstream& in) override;
```

- `saveData`:
  - In debug mode, first calls `applyPendingUpdates()` to write accumulated timestamp changes back into the `.lang` files.
  - Saves `missingKeys_` to the output stream via `missingKeys_.save(out)`.
- `loadData`:
  - Restores `missingKeys_` from the input stream.

This mechanism ensures that missing translations observed at runtime are periodically persisted to `missing_translations.txt` and that, in debug mode, language files are automatically updated with timestamps on save.

Debug Mode Support (QAPLA_DEBUG_I18N)

When `QAPLA_DEBUG_I18N` is defined, additional functionality is compiled in to ease translation development.

getI18nSourceDirectory()

```
std::filesystem::path getI18nSourceDirectory() const;
```

- Tries to locate the source `i18n` directory relative to the current working directory.
- Probes several candidate paths (e.g. project root, `build/`, `build/default/`) and checks for the presence of `eng.lang`.
- Returns the canonical directory path once a match is found.
- If no valid directory is found, logs a warning and falls back to `current_path() / "i18n"`.

markTimestampUpdate / applyPendingUpdates

```
void markTimestampUpdate(const std::string& langCode,
                         const std::string& topic,
                         const std::string& lookupKey,
                         const std::string& normalizedKey,
                         bool isOldFormat);

void applyPendingUpdates();
```

- `markTimestampUpdate`:
  - Records a `TimestampUpdate` entry for a `(language, topic, lookupKey)` triplet if not already present.
  - The `normalizedKey` is used when converting from old key formats.
- `applyPendingUpdates`:

- For each language with pending updates:
  - Opens the corresponding `<langCode>.lang` file from the source `i18n` directory.
  - Loads its sections using `QaplaHelpers::IniFile::load`.
  - For each `TimestampUpdate`:
    - Locates the `Translation` section for the matching topic.
    - Parses each key using `TranslationKey::parseKeyString`.
    - If an entry uses the **old format** (no timestamp):
      - Constructs a new key string with the current date using `TranslationKey` and replaces the key.
    - If an entry already uses the **new format** but has an outdated timestamp:
      - Updates the timestamp portion in-place while preserving the rest of the serialized key.
  - If any modifications were made, writes the updated sections back to the `.lang` file using `QaplaHelpers::IniFile::saveSections`.
- Clears `pendingUpdates_` afterwards.

This mechanism allows the system to automatically touch translation keys when they are actually used, keeping timestamps up to date without manual editing.

addMissingTranslationToFile()

```
void addMissingTranslationToFile(const std::filesystem::path& filepath,
                                 const std::string& topic,
                                 const std::string& key,
                                 const std::string& value);
```

- Adds new translation entries directly into a `.lang` file.
- Steps:
  1. Verifies that `filepath` exists; if not, logs a warning and returns.
  2. Loads all sections from the file.
  3. Uses `TranslationKey::parseKeyString` to obtain the `lookupKey` from the given `key` (handles old/new formats).
  4. Searches for the `Translation` section with a matching `id` (the topic).
  5. If found:
     - Checks whether any existing entry has the same `lookupKey` (in any format) and returns early if so.
     - Otherwise appends the new `key=value` pair.
  6. If no matching section exists:
     - Creates a new `Translation` section with `id=<topic>` and adds the entry.
  7. Writes all sections back to the file.

This is used exclusively in debug mode to fill in missing entries while the application is running.

Global Helper Function: tr()

```
inline std::string tr(const std::string& topic,
                      const std::string& text);
```

- Convenience wrapper for `Translator::instance().translate(topic, text)`.
- Intended as the main API for UI code:
  - `tr("Button", "OK")`
  - `tr("Tab", "Engines")`
- Respects the current language, debug/release behavior, and missing-key handling.

Usage Guidelines

- Always call `tr(topic, key)` in UI code instead of hard-coding user-visible strings.
- Use stable keys (typically English text) so that `TranslationKey` and `TranslationNormalizer` can produce consistent lookup keys.
- Do not store or reuse untranslated keys as if they were translated values; the i18n system expects the original English text as input for `translate()`.
- In debug builds:
  - Run the GUI with `QAPLA_DEBUG_I18N` enabled when working on localization.
  - Inspect the `.lang` files under `i18n/` – new keys and timestamps are added automatically.
- In release builds:
  - Missing translations are collected in `missing_translations.txt` in the configuration directory.
  - Use this file to update `.lang` resources for the next release.

Summary

The `Translator` class encapsulates all logic for language selection, translation lookup, and integration with both embedded resources and editable `.lang` files. The combination of `Autosavable`, `ConfigData`, and the debug hooks enables a workflow where missing translations are automatically discovered and persisted, while the normal runtime cost of translation lookups remains low and thread-safe.